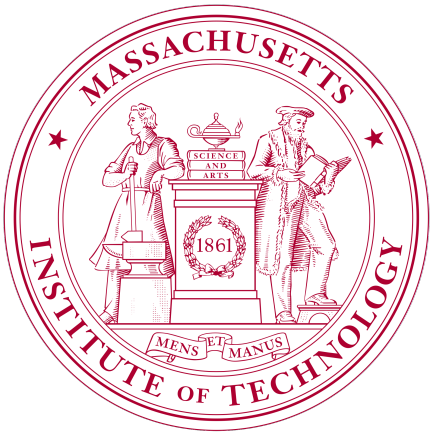
The spatial digital equalizer

6.111 Final Project

Alexander Sludds and Priya Kikani

December 2016



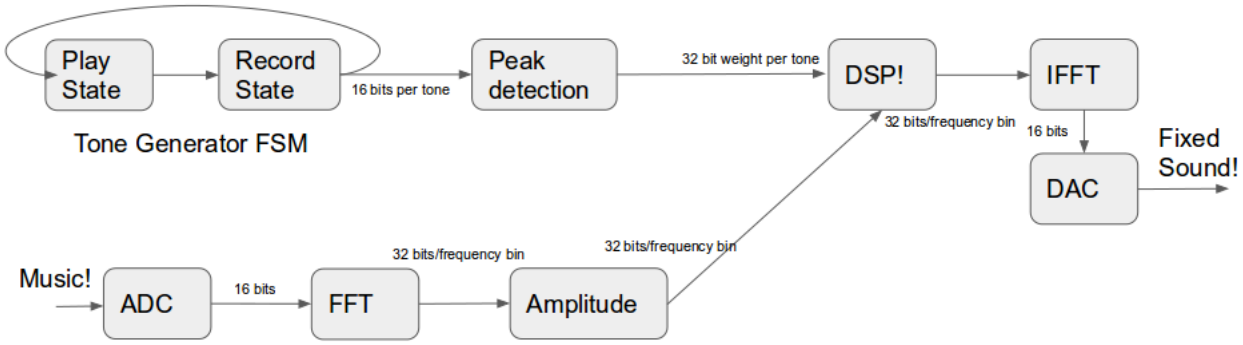
Contents

Introduction

While the sculpted chambers of Carnegie Hall offer an amazing musical experience, small rooms muffle and distort the grandeur of any listening experience. The acoustical properties of the room, such as its size and the material composition of its walls, directly impact the musical experience. Our goal is to demonstrate that by using spatial acoustical characterization and digital signal processing to adaptively equalize recorded music, sound quality will become independent of the environment's intrinsic acoustical characteristics. First, a frequency sweep will be transmitted into the room. Next, the reflected signal carrying acoustical information about the room will be collected. An FPGA will be utilized to generate a transfer function of the space. This data will be processed–also using the FPGA–to calibrate audio filters. These filters will pre-process the music with frequency-dependent compensation for the specific acoustic characteristics of the room. This project will provide users with concert-hall-quality music within virtually any space they choose.

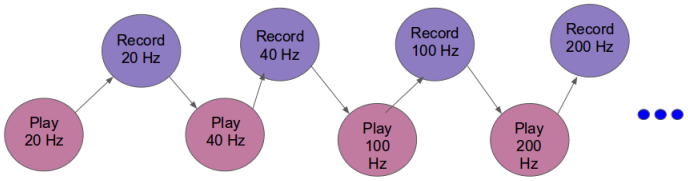
High Level Technical Overview

The overall project is divided into two separate stages. The first is room calibration (done by Priya). This stage characterizes the acoustics of the room and generates frequency dependent compensation values. The second stage is Music Adjustment (done by Alex). This stage applies the compensation values to the music and outputs the audio such that the music is pre-adjusted for the room.



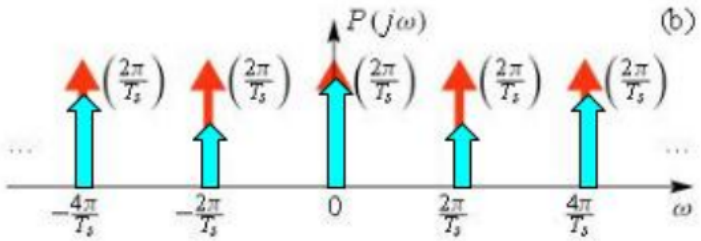
Play-Record FSM

(By Priya). This is a high level description of the Play/Record FSM. A series of tones are played at increasing frequencies corresponding to the audio perception abilities of the human ear. Those signals are then recorded and stored in memory. The relative amplitude of the recorded signals demonstrates the behavior of the room in the frequency domain–for example, if the recorded signals have attenuated high frequencies, then the room acts as a low pass filter. The user then has the option to play the values from memory. The peak detection module will then access the memory values in order to create compensation values for each frequency.



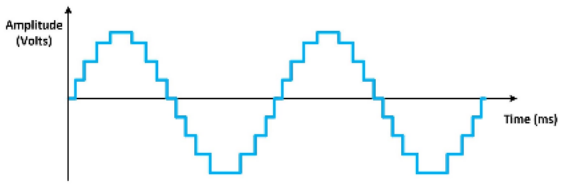
Peak Detection

(By Priya). For each tone, the peak detection module picks several memory values and averages them to generate a compensation value. This compensation value is then sent to the DSP module in order to adjust the filter weights.



ADC / DAC

(By Alex) In order to properly process audio we must be able to convert the analog audio signal that a device outputs into a digital audio signal that an FPGA can manipulate. This is done using an analog to digital converter (ADC). The ADC samples the analog waveform and approximates the analog voltage to the closest possible digital voltage, as is described by this sine wave:



The XADC chip available on the Nexys 4 board is a 12-bit 1MSPS analog to digital converter. The ADC is biased using a circuit that we shall describe later, but roughly we bias the input voltage halfway between the maximum and minimum input voltages so that we can maximize the maximum voltage swing. In order to allow the ADC to run at its fastest possible sampling rate we must supply a 104MHz clock to it which is taken care of by the clock wizard IP. This ADC signal is then passed onto the FIR.

The digital to analog converter (DAC) is implemented after all forms of digital signal processing. It converts our modified digital signal into an analog signal that can be played through a speaker.

FIR Filtering

(By Alex) FIR filtering stands for finite impulse response. When a signal is convolved with an FIR filter their frequency domain representations are multiplied. This is useful because an FIR filter's frequency domain response is easy enough to manipulate.

DSP and Sexy Presets

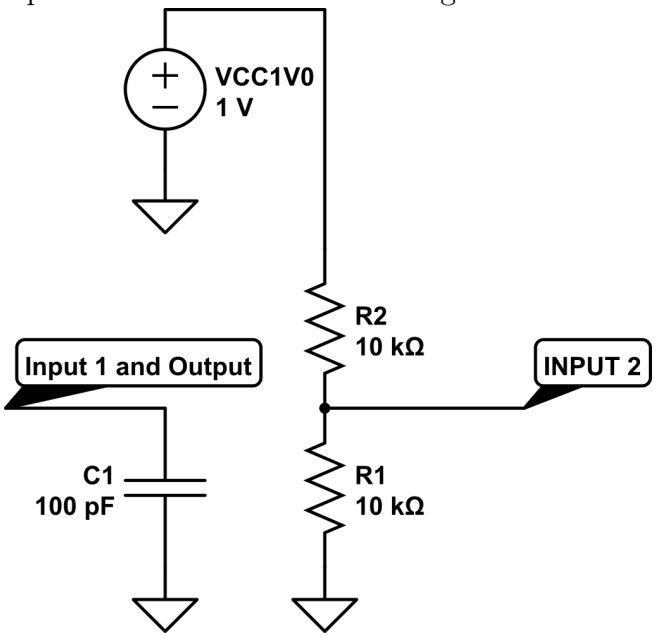
The DSP (Digital Signal Processing) module acts as a way for us to change the way that audio is played. For example, if we want to increase the base on some music we could change some switches on the nexys 4 board and that would amplify certain frequencies. The DSP module will be hardcoded with values that are generated by peak detection module for different frequency bins.

Audio Processing

(By Alex) As was noted on the high level block diagram there are two tracts required to implement this project, the creation of a transfer function before the DSP module and the processing of audio. This section covers audio processing.

ADC

The ADC is the onboard XADC module. This module samples at 1MSPS with 12 bits of resolution. The ADC requires a biasing circuit as it samples between 0V and 1V. The 0V and 1V supply are located on the nexys4 board. In order to maximize the range that the input voltage can swing over we shall bias the input at 0.5V. This is the biasing circuit:



In this circuit we connect the two leads of the 3.5mm jack to input 1 and 2. Input 2's DC bias is suddenly raised to 0.5V. Now input 1, as a result, will have a bias of 0.5V and passes an AC signal. We measure our output from input 1.

This bias circuit was taken from the "looper" demo available on the re-

source center for the nexys 4 board.

It must be noted that early on there were several issues with the bias to the ADC as it created a large amount of noise. Later this problem would be solved in verilog using oversampling.

The ADC takes this 12 bit signal and pads it with four zeros on the right side so that it is now 16 bits.

FIR

The FIR filter that we are using are 4000 tap filters. The reason that we use 4000 tap filers is because of a tradeoff, 500 filters was experimentally determined to be the best amount for meeting timing requirements and something larger than 4000 would provide high quality filtering (this was not tested because of the timing constraints on the FPGA). In the end I settled on 4000 taps as it provides very high quality filtering while still coming close to meeting the timing specifications.
Since the transfer function of the room is characterized at the following frequencies: 80Hz, 100Hz, 200Hz, 300Hz, 500Hz, 800Hz, 1000Hz, 1500Hz, 2000Hz, 3000Hz, 5000Hz, 8000Hz. We are going to choose the cutoff frequencies that we shall be using as the geometric mean between two frequencies.This is true except for the lower and upper frequencies where we shall use a low pass and high pass filter respectively.

The following frequencies are our cutoff frequencies: 90Hz , 140Hz, 245Hz, 390Hz, 630Hz, 900Hz, 1225Hz, 1730Hz, 2450Hz, 3875Hz, 6325Hz

When measured experimentally all frequencies bin cutoffs were within 5 percent of there expected value.

The FIR IP Core module takes the padded 16 bit signal from the ADC and then runs each of these FIR filters in parallel. Since they are run in parallel the different frequencies can all give their outputs in the time domain at the same time. This means that after filtering for specific frequency bands we recombine them together in order to obtain our final signal.

The fir filters themselves are generated in matlab according to the following lines of code:

For the lowest frequency we need only consider a low pass filter
$b = fir1(4000, \frac{\omega_c}{\frac{\omega_s}{2}})$;
$final = round(b * 2^{22})$;

For mid-range frequencies we shall consider bandpass filters:
$b = fir1(4000, [\frac{\omega_{c1}}{\frac{\omega_s}{2}} \frac{\omega_{c2}}{\frac{\omega_s}{2}}])$;
$final = round(b * 2^{22})$;

For the highest frequency we will also be using a bandpass filter up to an inaudible range:
$b = fir1(4000, [\frac{\omega_{c1}}{\frac{\omega_s}{2}} \frac{\omega_{c2}}{\frac{\omega_s}{2}}])$;
$final = round(b * 2^{22})$;

Please note that the value 2^{22} was experimentally determined.

I used the following python code to convert matlab vectors into a usable format.

```
bin1= "matlab vectors go here"
bin2= "more numbers here"
bin3= "there were 12 bins"
def FIRtolist(matlabstring):
    matlabstring = matlabstring.replace('\t',",")
    matlablist = matlabstring.split(',')
    matlablist = [int(x) for x in matlablist]
    return matlablist

print FIRtolist(bin1)
print "\n"
print "above is bin1"
raw_input("Press Enter to continue...")
print FIRtolist(bin2)
print "\n"
print "above is bin2"
raw_input("Press Enter to continue...")
print FIRtolist(bin3)
print "\n"
print "above is bin3"
print "This pattern continues until bin12"
```

DSP

The DSP module is a case statement over the different filter options (EDM, bass, treble) which changes the output value by choosing which bits we select. For example, if the EDM button is being pressed we wish to increase the amplitude of low and high frequencies. Thus, we shift the midrange frequencies down (right shift by 1). The reason why we don't shift the low and high frequencies up is because if we do this then we can corrupt the audio(this was verified experimentally).
During testing this section was not functioning because the large amount of FIR taps meant that the timing specification was not being met. However, on the 500 tap, when the timing spec was a lot closer to being met, the DSP module worked well, with a small amount of noise being added. The problem was that the edges of the filter dropped off very slowly so different frequencies bins would interfer with each other.
It is also within this module that the transfer function's values are implemented. They are stored as constants and can be changed, but the project must be recompiled. By default the values are 6 bits and we choose to "bias" the transfer function value around $\frac{2^6}{2} = 32$. At the time of checkoff these values were slightly altered (increased or decreased by 1) based upon a transfer function that Priya gave me.
At one point I spent about two hours changing the different clocking frequencies of various FIR and DSP components in order to see if it would increase the performance of the overall system (meet timing spec better or improve the FIR filter). However, changing the clock frequency does not change the performance.

DAC

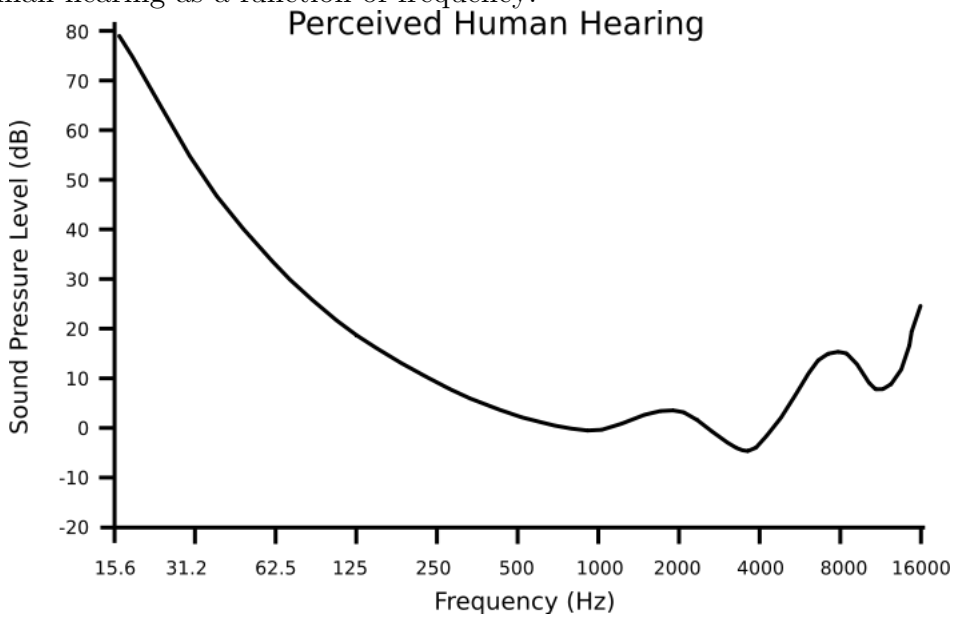
The DAC that we are using outputs directly into the mono audio output. It uses a PWM signal with an 11 bit output. The reason that the output has so few bits is because we are clocking the module at 104mHz so the highest possible signal that we can represent is $1.04 * 10^8/2^{11} = 51kHz$.
The code that I used to power this module was originally written by Mitchell and I did not change it.

Transfer Function Generation

(By Priya). The specific details of transfer function generation are given here.

First, various tones are generated via a method similar to that introduced in lab5. An AC97 Audio Codec chip is used to interface between the analog speakers and digital tone generation. The AC97 chip can both accept audio input from a microphone as well as output audio data to a speaker. Both functions were used for this project.

The human ear can perceive audio from 20 Hz-20,000Hz. However, the human ear can perceive and differentiate between lower frequencies much better than higher frequencies. The figure below demonstrates perceived human hearing as a function of frequency:



As a result, the following frequencies were used in the final setup: 80Hz, 100Hz, 200Hz, 300Hz, 500Hz, 800Hz, 1000Hz, 1500Hz, 2000Hz, 3000Hz, 5000Hz, 8000Hz. The microphone used had a frequency range from 80Hz to 12,000Hz. Various tones with frequencies up to 15,000Hz were tested, though it was found that it was difficult to perceive sound above 8,000Hz.

In order to generate these tones, the AC97 will take pulse-code modulated (PCM) data and send it to two DACs, which will then output two 48kHz analog waveforms, one for each stereo speaker. Because the AC97

chip expects data at a rate of 48kHz, PCM data is driven with coefficients that correspond to a sine wave at each frequency. For example, a 80 Hz tone is generated with 48khz/80hz=600 coefficients per 48khz clock cycle. The coefficients have a width of 12 bits in order to allow for enough resolution but also not consume too much space in memory. The specific coefficients for each frequency were generated with the following MATLAB code (in this case for an 80 hz tone) :

```
% Make 480000 samples in the range 0 to 1 second
t = linspace(0, 1, 48000);

% Assign signal charachteristics (period, amplitude, and phase shift)
period = 1/80; % x hertz = period of 1/x of a second.

A=4095; %Amplitude for 12 bits
phaseShift = 0 * pi/180; % In radians.

x= sin(2*pi*t/period - phaseShift);
%Test signal (expects signal to be between -1 and 1)
sound(x, 48000);

%Define signal for Verilog
y = A*sin(2*pi*t/period - phaseShift);

y=y';

out=round(y);

%cycle length
T=round(48000/(1/period));
for k=1:T/2
    Y=['10''d',num2str(k-1),': pcm_data <= 12''sd',num2str(out(k)),';']
    disp(Y);

end

for m=1:T/2
    X=['10''d',num2str(T/2+m-1),': pcm_data <= -12''sd',
```

```
        num2str(abs(out(T/2+(m)))),';'];
    disp(X);
end
plot(t,y,'b');
```

This MATLAB code also formats the generated coefficients for translation to Verilog. The Verilog operates by first creating an "index" register that increments at a rate of 48 khz, though the overall clock for this module operates at a 27 mhz rate. "Index" then functions as the case of a case statement, with new coefficients from the MATLAB code driven to PCM data each time index increments. This Verilog module ensures that the coefficients of each tone is appropriately mapped to the 48Khz clock necessitated by the AC97 chip. Each tone had a separate verilog module which contained the specific coefficients. An example module is given in the Source Files.

One issue with this approach is that the AC97 chip introduced a lot of noise into the low frequency tones. Using the coefficients to generate and play a tone via MATLAB produced the expected sine wave and pure tone; however, the output of the AC97 with those same coefficients did not look like a pure sine wave when measured with an oscilloscope. This effect was not at all evident at frequencies above 200 Hz.

Each tone was played for one second each (this was dictated by the Play/Record FSM). The next step to generate a transfer function of the room is to record the tones in memory. The purpose behind this step is that the played tones will travel through the room and reflect off its walls; therefore, the recorded signals will carry information about the room.

The AC97 chip also takes incoming data from the microphone. It is passed through an ADC, which samples the analog waveform at a 48 kHz rate and digitizes the waveform to an 18-bit value. To save memory, only the higher order 12 bits are used (In contrast, Lab 5 only used the higher order 8 bits; this projected aimed to record with higher resolution). The signals are recorded in memory with a mybram module for each of the twelve tones. Rather than storing every value from the AC97 in memory (and thereby sampling at 48khz), specific low frequencies were down sampled to 6 khz. Intermediate frequencies were sampled at a 12 khz rate, and the highest frequency was sampled at a 48 khz rate.

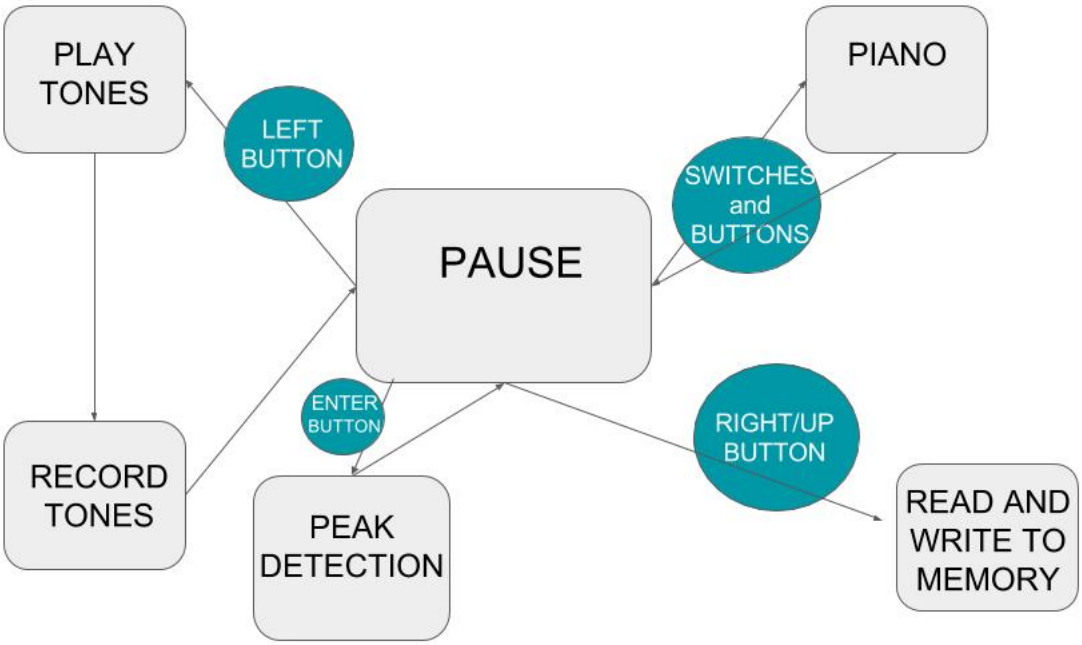
The third step in the room characterization process is to read those samples from memory and generate a single value describing the magnitude of the record signal. Lastly, those values must be displayed for use later.

Play-Record FSM

The Play/Record FSM is the mastermind behind the room characterization. It has a default state of PAUSE, which allows the user to employ the different functionalities of the project via the labkit buttons and switches, as shown in the below diagram. Because each tone had a separate state for each functionality (Play, Record, etc) the final FSM had 32 states. In retrospect, the complexity of the FSM could have been reduced by combining states and perform computations in parallel, though this approach allowed for an easily modifiable design.

In order to play the different tones, a counter based off the underlying 27 mhz clock dictates that each tone is played for 1 second, as each state will send a different audio output to the AC97. The record tone states write to memory with specific down sampling rates for each frequency.

Before the peak detection step, the FSM transitions to the PAUSE state. This allows the user to play piano with the labkit switches and buttons before undergoing peak detection.



Peak Detection

The peak detection section is essential for converting the recorded signal to a single magnitude value. Two approached were explored as potential peak detection methods. The first approach summed over each tone in memory and scaled to prevent overflow.

The second approach is to use the prior information of each frequency and memory size. Because those two numbers are well known, it is possible to calculate the expected locations in memory where the peak of the recorded signal should be located.

After experimentation, it was evident that the first approach yielded more accurate results; this also allows the peaks to be computed during the recording process rather than in a separate state.

Additional Features

This section describes the various stretch goals implemented in this project.

Display Transfer Function

(By Priya). Two different techniques were employed to display the transfer function. First, the exact values of each peak were displayed on the LEDs. The 12 bit number was expressed as 3 hex values, and the value corresponding to each tone could be accessed via flipping the labkit switch or button corresponding to each tone.

The second display technique was to display color bars on the monitor which expressed the comparative peaks. Each tone has an individual color bar where the length corresponds to the calculated magnitude of that tone.

One challenge in this stage arose from dealing with multiple clock domains. In order to interface with the VGA display, the display code required a 65 mhz clock. However, the audio processing was done at a 27 mhz clock. As a result, the values dictating the length of each color bar were given to the display code too slowly, which meant that the direct output from the peak detection couldn't be displayed in real time.

To fix this problem, the display code was modified such that the display module expected a new value from the Play/Record FSM at a rate of once

per frame, rather than once per 65 mhz clock cycle. This allowed the color bars to display, and it also resulted in the interesting side effect of allowing the user to witness the peak detection calculation in real time.

Save Different Transfer Functions

(By Priya) After the transfer function is displayed on the monitor, it was relatively straightforward to allow the user to save different coefficients in memory and display them at will. For example, to save one set of coefficients in memory, 12 12-bit registers were instantiated within the display tones module. When the user depresses the "UP" button, the current values are loaded into the registers. In order to save multiple coefficients, different sets of registers were instantiated, and the user is able to write to each one based on the lab switches. Reading from memory works similarly, with the "RIGHT" button corresponding to reading from memory, with prior saved coefficients being displayed on the monitor. The user has the options to save three different sets of coefficients. Due to this implementation, the coefficients are not saved if the labkit is reprogrammed or turned off.

Lab Switch Piano!

(By Priya) The 12 different tones were wired to the labkit switches and buttons, with switch 0 corresponding to 80 hz, switch 1 corresponding to 100 hz, etc. In the PAUSE state the user is able to flip those switches to play piano. Furthermore, specific chords can be played by enabling multiple switches. In this scenario, the output of the multiple tone generators is added and shifted right to prevent overflow. The resulting sound is very distinct from the individual switches. This feature was fun to implement as well as useful for debugging.

Additional DSP Presets

(By Alex) A goal of the project was to add additional presets such as bass, EDM, and treble. This was accomplished using bit shifts. When we are summing together the fir filtered audio was are dealing with large-bit signals (40 bits) and as a result just take the top 32 bits. When we sum together the 12 frequency bins we then shift the sum right 4 bits. In order to do DSP for the presets we have our DSP case statement choose different bit values

for certain frequency bins. For example, originally the sum was calculated using this line of verilog:

```
superaudio <=(((fbin1[36:5]>>5)*transfer1+(fbin2[37:6]>>5)*transfer2
+ (fbin3[37:6]>>5)*transfer3 + (fbin4[37:6]>>5)*transfer4
+ (fbin5[37:6]>>5)*transfer5 + (fbin6[37:6]>>5)*transfer6 +
(fbin7[37:6]>>5)*transfer7 + (fbin8[37:6]>>5)*transfer8 +
(fbin9[37:6]>>5)*transfer9 + (fbin10[37:6]>>5)*transfer10 +
(fbin11[38:7]>>5)*transfer11 + (fbin12[38:7]>>5)*transfer12)>>4);
```

However, for EDM we output this line of verilog:

```
superaudio <=(((fbin1[36:5]>>5)*transfer1+(fbin2[37:6]>>5)*transfer2
+ (fbin3[37:6]>>5)*transfer3 + (fbin4[37:6]>>5)*transfer4 +
(fbin5[37:6]>>6)*transfer5 + (fbin6[37:6]>>6)*transfer6 +
(fbin7[37:6]>>6)*transfer7 + (fbin8[37:6]>>5)*transfer8 +
(fbin9[37:6]>>5)*transfer9 + (fbin10[37:6]>>5)*transfer10 +
(fbin11[38:7]>>5)*transfer11 + (fbin12[38:7]>>5)*transfer12)>>4);
```

Note that for EDM we increased the amount that we right shifted the central frequencies and left the bass and treble frequencies alone.

Equipment Testing

Test Multiple Speakers

(By Priya and Alex). Initially, for the sake of debugging, a pair of standard (non-wireless) apple headphones were used to debug the sound of the project. When we were ready to test in larger areas we used a "good" speaker and a "bad" speaker. Here "good" means a speaker which we believe to be better because of the quality of the manufacturing and also the size of the speaker (smaller speakers preform worse at lower frequencies because they have a smaller resonant cavity). It was shown in the project that different speakers produced different transfer functions.

Test Microphone

(By Priya and Alex) We used a single microphone for the project. The microphone did a reasonable job, though the measured frequency response of the microphone was limiting (the lower bound for measuring was 80Hz and upper bound was 12,000 Hz).

Personal Reflections

Alex's Reflection

Though all aspects of the project that we wanted to implemented were implemented some of them were poor as a result either hardware limitations or experimentally determining values. Some things were unavoidable (sexy presets not working as expected) but some worked substantially better than I expected (summing all FIR filters together to recreate the audio).
In terms of time put into the project I wasted several days trying to figure out how to best transform Mitchell's FFT code into an "audio pipeline" that takes audio into, takes an FFT, passes that into an IFFT, and then puts the original audio on the output. As of right now I am still unclear as to why my original approach did not work for the IFFT, it could have possibly been an issue with buffering the output data of the IFFT.
I spent a very long amount of time grappling with the analog audio input into the FPGA. Not that this was a difficult part, but some of the connections would occasionally come loose. I fixed the issue by soldering the 3.5mm jack's leads to some leads coming out of the breadboard. However, this added a noticeable amount of noise to the signal. I spent several hours trying some different techniques in order to remove the noise, but none of them were successful. What did work was oversampling the audio 16 samples at a time.
Other than small gripes with my approach to the project I think this was an incredibly rewarding experience. My advice to future students who are starting their project would be that starting early and picking the brains of engineers are incredibly valuable things to do. I had some of my best breakthrough moments because I talked to people working on similar projects and that gave advice on what did and didn't work for them.
On the last two days of the project, after all of the infrastructure had been put in place, all of my time went into creating FIR filter coefficients and putting them in the correct formatting.

Priya's Reflection

I enjoyed working on this project, and it was amazing to see everything come together. I think that with a few more days I could have implemented real

time transfer function generation. A lot of time went into generating the specific tones and trying to make them sound good.

Because 12 tones were used, there was a lot of repeated tasks. As in, the code for playing the tones, recording the tones, calculating the peaks, and displaying and saving that information had to be replicated 12 times. If I was to redo this project, I would spend more time coming up with more elegant means of performing these computations.

Summary

In conclusion, this project was able to meet stated expectations. Further work includes electronic transfer of room data to audio processing as well as real time implementation of transfer function generation. We would like to extend our thanks to Gim Hom for his guidance and teaching throughout the semester. Also, Joe Steinmeyer was instrumental in debugging critical parts of the project. Mitchell, Alex S. and Valerie were also amazingly helpful.

Source Files

Example Verilog Module for Tone Generation

```
module tone500hz(
    input clock,
    input ready,
    output reg signed [11:0] pcm_data
    );
 reg [6:0] index;

   initial begin
      index <= 7'd0;
      // synthesis attribute init of index is "00";
      pcm_data <= 12'd0;
      // synthesis attribute init of pcm_data is "00000";
   end

   always @(posedge clock) begin
      if (ready) begin
                        if(index <7'd95) begin
                                index <= index+1;

                            end
                            else if(index >=7'd95) begin
                                index <=0;
                            end

                end
   end

   // one cycle of a sinewave in 64 20-bit samples
   always @(index) begin
      case (index[6:0])
        7'd0: pcm_data <= 12'sd0;
7'd1: pcm_data <= 12'sd134;
7'd2: pcm_data <= 12'sd267;
7'd3: pcm_data <= 12'sd399;
```

```
7'd4: pcm_data <= 12'sd529;
7'd5: pcm_data <= 12'sd657;
7'd6: pcm_data <= 12'sd783;
7'd7: pcm_data <= 12'sd904;
7'd8: pcm_data <= 12'sd1023;
7'd9: pcm_data <= 12'sd1136;
7'd10: pcm_data <= 12'sd1245;
7'd11: pcm_data <= 12'sd1348;
7'd12: pcm_data <= 12'sd1446;
7'd13: pcm_data <= 12'sd1538;
7'd14: pcm_data <= 12'sd1622;
7'd15: pcm_data <= 12'sd1700;
7'd16: pcm_data <= 12'sd1771;
7'd17: pcm_data <= 12'sd1834;
7'd18: pcm_data <= 12'sd1889;
7'd19: pcm_data <= 12'sd1936;
7'd20: pcm_data <= 12'sd1975;
7'd21: pcm_data <= 12'sd2006;
7'd22: pcm_data <= 12'sd2028;
7'd23: pcm_data <= 12'sd2041;
7'd24: pcm_data <= 12'sd2045;
7'd25: pcm_data <= 12'sd2041;
7'd26: pcm_data <= 12'sd2027;
7'd27: pcm_data <= 12'sd2006;
7'd28: pcm_data <= 12'sd1975;
7'd29: pcm_data <= 12'sd1936;
7'd30: pcm_data <= 12'sd1889;
7'd31: pcm_data <= 12'sd1834;
7'd32: pcm_data <= 12'sd1771;
7'd33: pcm_data <= 12'sd1700;
7'd34: pcm_data <= 12'sd1622;
7'd35: pcm_data <= 12'sd1537;
7'd36: pcm_data <= 12'sd1446;
7'd37: pcm_data <= 12'sd1348;
7'd38: pcm_data <= 12'sd1245;
7'd39: pcm_data <= 12'sd1136;
7'd40: pcm_data <= 12'sd1022;
7'd41: pcm_data <= 12'sd904;
```

```
7'd42: pcm_data <= 12'sd782;
7'd43: pcm_data <= 12'sd657;
7'd44: pcm_data <= 12'sd529;
7'd45: pcm_data <= 12'sd399;
7'd46: pcm_data <= 12'sd267;
7'd47: pcm_data <= 12'sd134;
7'd48: pcm_data <= -12'sd0;
7'd49: pcm_data <= -12'sd134;
7'd50: pcm_data <= -12'sd267;
7'd51: pcm_data <= -12'sd399;
7'd52: pcm_data <= -12'sd529;
7'd53: pcm_data <= -12'sd657;
7'd54: pcm_data <= -12'sd783;
7'd55: pcm_data <= -12'sd905;
7'd56: pcm_data <= -12'sd1023;
7'd57: pcm_data <= -12'sd1136;
7'd58: pcm_data <= -12'sd1245;
7'd59: pcm_data <= -12'sd1348;
7'd60: pcm_data <= -12'sd1446;
7'd61: pcm_data <= -12'sd1538;
7'd62: pcm_data <= -12'sd1623;
7'd63: pcm_data <= -12'sd1700;
7'd64: pcm_data <= -12'sd1771;
7'd65: pcm_data <= -12'sd1834;
7'd66: pcm_data <= -12'sd1889;
7'd67: pcm_data <= -12'sd1937;
7'd68: pcm_data <= -12'sd1975;
7'd69: pcm_data <= -12'sd2006;
7'd70: pcm_data <= -12'sd2028;
7'd71: pcm_data <= -12'sd2041;
7'd72: pcm_data <= -12'sd2045;
7'd73: pcm_data <= -12'sd2041;
7'd74: pcm_data <= -12'sd2027;
7'd75: pcm_data <= -12'sd2006;
7'd76: pcm_data <= -12'sd1975;
7'd77: pcm_data <= -12'sd1936;
7'd78: pcm_data <= -12'sd1889;
7'd79: pcm_data <= -12'sd1834;
```

```
7'd80: pcm_data <= -12'sd1771;
7'd81: pcm_data <= -12'sd1700;
7'd82: pcm_data <= -12'sd1622;
7'd83: pcm_data <= -12'sd1537;
7'd84: pcm_data <= -12'sd1446;
7'd85: pcm_data <= -12'sd1348;
7'd86: pcm_data <= -12'sd1245;
7'd87: pcm_data <= -12'sd1136;
7'd88: pcm_data <= -12'sd1022;
7'd89: pcm_data <= -12'sd904;
7'd90: pcm_data <= -12'sd782;
7'd91: pcm_data <= -12'sd657;
7'd92: pcm_data <= -12'sd529;
7'd93: pcm_data <= -12'sd399;
7'd94: pcm_data <= -12'sd267;
7'd95: pcm_data <= -12'sd133;


        endcase // case(index[5:0])
    end // always @ (index)

endmodule
```